IL 00/00516

PCT/IL 00/00516
12 OCTOBER 2000

REC'D 27 OCT 2000
WIPO PCT

PA 306481

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE

United States Patent and Trademark Office

#2
10/069234

September 28, 2000

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A FILING DATE UNDER 35 USC 111.

APPLICATION NUMBER: *60/151,795*
FILING DATE: *August 31, 1999*

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

PATENT AND TRADEMARK OFFICE · UNITED STATES OF AMERICA

By Authority of the
COMMISSIONER OF PATENTS AND TRADEMARKS

N. Williams

N. WILLIAMS
Certifying Officer

PROVISIONAL PATENT APPLICATION COVER SHEET

This is a request for filing a PROVISIONAL PATENT APPLICATION under 37 C.F.R. §1.53(c).

Docket Number	66430

INVENTOR(S)/APPLICANT(S)

LAST NAME	FIRST NAME	MIDDLE INITIAL	RESIDENCE (CITY AND EITHER STATE OR FOREIGN COUNTRY)
1. Shmueli	Oded		Nofit, Israel
2. Konopnicki	David		Haifa, Israel
3. Leiba	Lior		Haifa, Israel
4. Sagiv	Yehoshua		Jerusalem, Israel

TITLE OF THE INVENTION

A FORMAL YET PRACTICAL APPROACH TO ELECTRONIC COMMERCE

CORRESPONDENCE ADDRESS

FITCH, EVEN, TABIN & FLANNERY
Suite 1600
120 South LaSalle Street
Chicago, Illinois 60603-3406
Telephone: (312) 577-7000
Facsimile: (312) 577-7007

CUSTOMER NUMBER 22242

22242

STATE	IL	ZIP CODE	60603-3406	COUNTRY	USA

ENCLOSED APPLICATION PARTS (check all that apply)

[X] Specification	Number of Pages: 20	[ ] Small Entity Statement	
[ ] Drawing(s)	Number of Sheets: 0	[ ] Other (specify)	

METHOD OF PAYMENT OF FILING FEES FOR THIS PROVISIONAL PATENT APPLICATION (check one)

	Provisional Filing Fee Amount(s)	$ 150.00 -------- (less one-half if small entity) = $75.00
[X] A check or money order is enclosed to cover the Provisional Patent Application filing fees		
[ ] The Commissioner is hereby authorized to charge any deficiencies in filing fees, or credit overpayments, to Deposit Account Number: 06-1135		

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

[X] No.

[ ] Yes, the name of the U.S. Government agency and the Government contract number are: ____________

Respectfully submitted,

SIGNATURE Robert J. Fox

Date August 31, 1999

TYPED or PRINTED NAME Robert J. Fox

REGISTRATION NO. 27,635
(if appropriate)

[ ] Additional inventors/applicants are being named on separately numbered sheets attached hereto.

PROVISIONAL PATENT APPLICATION FILING ONLY

Provisional Cover Sheet 6-99 p.1/1

A/prov

JC604 U.S. PTO 08/31/99

PROVISIONAL PATENT APPLICATION TRANSMITTAL
(37 C.F.R. §1.53(c))

Attorney Docket No. 66430

JC583 U.S. PTO 60/151795 08/31/99

Box PROVISIONAL PATENT APPLICATION
Commissioner of Patents and Trademarks
ATTENTION: Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

Transmitted herewith for filing under 37 C.F.R. §1.53(c) is the provisional patent application of:

Title: A FORMAL YET PRACTICAL APPROACH TO ELECTRONIC COMMERCE

CERTIFICATE OF MAILING BY "EXPRESS MAIL"

"Express Mail" Mailing Label Number EL 358 627 273 US

Date of Deposit August 31, 1999

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" Service under 37 CFR §1.10 on the date indicated above and is addressed to the Commissioner of Patents and Trademarks, Washington, D.C. 20231.

Ed Price
(Typed or printed name of person mailing)

Ed Price
(Signature of person mailing)

60151795 .083199

TRANSMITTAL LETTER FOR PROVISIONAL PATENT APPLICATION

Enclosed are:

(X) Cover Sheet for the above-identified provisional patent application identifying the application as a provisional application.

( ) Verified Statement Claiming Small Entity Status.

(X) A specification and 0 sheets of Figures (( ) Formal, ( ) Informal) for the provisional patent application, totalling 20 pages.

(X) A check in the amount of $ 150.00 to cover the filing fee for the above-identified provisional patent application without a claim of small entity status.

( ) A check in the amount of $ 75.00 to cover the filing fee for the above-identified provisional patent application by an entity claiming small entity status.

( ) Charge $__________ to Deposit Account No. 06-1135.

(X) A separate written request under 37 C.F.R. §1.136(a)(3) which is a general authorization to treat any concurrent or future reply requiring a petition for an extension of time under 37 C.F.R. §1.136(a) for its timely submission as incorporating a petition for an extension of time for the appropriate length of time therein.

(X) The Commissioner is hereby authorized to charge any additional fees which may be required in this application under 37 C.F.R. §§1.16-1.17 during its entire pendency, or credit any overpayment, toDeposit Account No. 06-1135. Should no proper payment be enclosed herewith, as by a check being in the wrong amount, unsigned, post-dated, otherwise improper or informal or even entirely missing, the Commissioner is authorized to charge the unpaid amount to Deposit Account No. 06-1135. This sheet is filed in triplicate.

(X) Address all future communications to Customer Number 22242.

22242

60151755.083199

August 31, 1999
(Date)

Robert J. Fox
Robert J. Fox
Registration No. 27,635

FITCH, EVEN, TABIN & FLANNERY
Suite 1600
120 South LaSalle Street
Chicago, Illinois 60603-3406
Telephone: (312) 577-7000
Facsimile: (312) 577-7007

A Formal Yet Practical Approach to Electronic Commerce

David Konopnicki, Lior Leiba, Oded Shmueli
Computer Science Dept.
Technion — Israel Institute of Technology
Haifa, Israel
{konop, lior, oshmu}@cs.technion.ac.il

Yehoshua Sagiv
Institute of Computer Science
The Hebrew University
Jerusalem, Israel
sagiv@cs.huji.ac.il

Abstract

This work explores (semi-)automated EC on the WWW. We outline a system in which trading parties present intentions, made of more elementary components, which are used to express their willingness to engage in deals subject to constraints. Parts of intentions may be variable components. Some variable components may be associated with computations that transform them into, usually, "more specified" components. This mechanism encodes business rules. By "fitting" intentions contracts are formed.

The EContracts frameworkis a specific realization of the architecture. enables EC WWW sites and EC automated tools to present standardized information. This information (1) allows each party to decide whether it wishes to engage in an EC activity with the other party, (2) enables automated negotiation between the parties, and (3) enables the establishment of an electronic contract, i.e., a formal description of an agreed upon EC transaction.

The EContracts framework defines the basic software components of an EC party and their interconnections. Based on the EContracts framework, various applications can be built. Examples are deal making applications, deal feasibility checkers, brokers etc. Furthermore, the definitions of the data structures and the algorithms enable a theoretical investigation of automated commerce.

1. Structuring Electronic Commerce

Electronic Commerce (EC) is expected to be the main activity on the WWW. Currently, EC is mainly browser-based. This stems primarily from reliance on "local" technology, lack of standards and non-uniform interfaces. Browser-based interaction wastes a lot of customers' time. In the near future, we expect a rich network-based marketplace, with hundreds if not thousands of major "players" and millions of smaller-scale "players." Furthermore, we expect almost everything to be tradable online, from consumer goods to insurance policies, from real estate parcels to software. In this kind of environment, browsing is a hopeless proposition. What is needed is a universal formalism ("the HTML of EC") that will support deal making on a global scale and protocols ("the HTTP of EC") that will enable the realization of automatic tools (agents). In this paper we outline a design to enable semi-automated EC on the WWW. By "semi-automatic," we leave open the possibility that a human decision maker takes part in the negotiation/decision phases of a commercial activity.

Our vision is that *parties* (a general term designating individuals, corporations, government entities, and other entities of legal standing) can specify *intentions*, a formal outline of deals in which such parties are ready to engage. Intentions are made of *components*. Components may be atomic or compound (to any depth). Furthermore, a component may be a *variable component*, that is unspecified (except perhaps for type information). Furthermore, components may be inter-related (e.g., by containment, by edge or labeled-edge connection, or by arbitrary predicates). An important facet of a variable component is its association with one or more computational devices (a generic term for hardware - including electronic and biological, software - programs, executables, byte-codes, systems, servers and the like). Such a computational device, based on its perceived state (which may comprise inputs, values of various components, values of certain other entities such as files, databases) and messages, transforms a variable component into a component. The "new" component is usually "more specific" than the variable it replaces. The idea is that such variable components and their associated computational devices embody "transient," "policy dependent" or "current availability" aspects of the willingness to engage in a deal. It is desirable, although not mandatory, that the functionality of the computational device be readily understood by inspection, a property we call *analyzability*.

Making a deal means reconciling the constraints placed on deals by the (two or more) parties involved. For simplicity, let us concentrate on two parties, the notions general-

ize to multi-party scenarios. By "reconciling" we mean "as best as possible subject to the parties' directives as well as more general laws that may apply". Looking at two intentions, we can think of reconciling the constraints as a form of "fitting" under constraints. Abstractly, this process "fits" the component structure of one party with the "counterpart" components of the other party. We envision a party as employing a computational entity, which we term "party machine (PM)," which controls the fitting of intentions, the PM may communicate with other computational devices, and in particular other PMs, in attaining its mission. For example, it may be responsible for activating the "fitting process" or activating the computational device associated with a variable component.

There are some very basic requirements for supporting a scenario as outlined above. First, a common terminology for intentions is needed. The analogy here is the natural language used by humans, suitably formalized, for commercial activities. This means that the intentions of a party be readily "understood" by other parties. Second, an agreed upon architecture is needed so that a PM of a party can assume certain abilities of an PM of another party. An analogy here is the client-server architecture of the WWW. Third, as stated, computational devices may issue messages and require response, this forms a foundation for automated, or semi-automated, negotiations. So, as we deal with automated tools, a protocol for negotiations need be established. Humans exercise such "protocols" all the time, although they are mostly unaware of this fact (e.g., recall your last visit to a fast food stand).

In designing solutions for the above mentioned three requirements we list some desirable properties. The common terminology should be simple, yet expressive and powerful. The architecture should be modular and "orthogonal," i.e., different modules should address different concerns. To enable a rich set of commerce modes, the structure and content of EC parties should be machine-analyzable. Machine analyzability will give rise to greater efficiency as well (see below). Of course, an EC party should be able to have "opaque" portions that are not viewable by other parties.

Using these concepts, various applications can be built. Examples are deal making applications, brokers, deal feasibility checkers, etc. Furthermore, the definitions of the data structures and algorithms enable a formal investigation of automated commerce.

The EContract framework presented in this paper is a step towards achieving the goals just outlined. It is a realization of the concepts outlined above. We should note, however, that other realizations of these concepts are possible. In the interest of concertinas, we have chosen for EContract a "logical flavor" along the lines of Prolog and logic programming. Other realizations may rely on LISP and functional programming, on more natural language oriented formalisms, on Java, C++ and Object Oriented formalisms and many more. By being concrete, we explore the promise of our approach and are able to formulate technical problems whose computational complexity may be evaluated.

The first step is to ensure that intentions are universally understood. For this we need a mechanism for forming intentions. In EContracts, a component is represented as a *rooted labeled tree*. In fact, an intention is also a rooted labeled tree which is composed of components, together with various constraints. The most basic components are *simple atomic entities*, e.g., of type integer, float, string. Next are *basic components* that are essentially (usually small) trees whose structure is agreed upon to represent a concept (e.g. car, buy, address). These basic components are called *classes* and they form the "words" of the common language. The word "class" hints at the fact that in an object oriented realization, these component are likely to be represented as (object oriented) classes. A component may be a *variable component*. In this case it appears as a single node labeled with a typed variable (its atomic type or its class name).

Using classes, the parties compose their intentions, essentially forming "sentences" which in turn define possible deals. As noted, the purpose of an intention is to describe a deal that a party is willing to engage in. For example, an intention can express that the *BooksOnline Corp. is selling books and that if you buy more than five books, you receive a 10% discount.* In EContracts, the mechanism that composes words into sentences (classes into intentions) relies on "variable instantiation" and the introduction of "operator nodes." A (leaf) variable component of an intention may be associated with a computation device, called a "commerce automaton" (CA) in this realization, which prescribes how the variable may be instantiated further during a later phase. A commerce automaton may outline a message exchange sequence between the parties. In addition to intentions, an EC party also maintains *party information*, a relational database containing information relevant to the party's activities. This is part of the "system state".

A deal is manifested by creating a mutually agreed upon *electronic contract* (EContract). The process of obtaining an EContract begins with two initial intentions, presented by the parties. A formal process, called *unification*, a part of the realization of "fitting", is used to construct an agreed upon EContract, provided such a contract is feasible. Unification may also be used by an EC party to determine whether an EContract is at all possible, prior to entering actual negotiations with the other party. Hence the importance of machine analyzability.

The EContract framework defines the basic software components of an EC party and their interconnections. The structure of an EC party is shown in Figure 1, the EContracts realization of a PM is:

- The Negotiation Control Program (NCP) is an overall co-

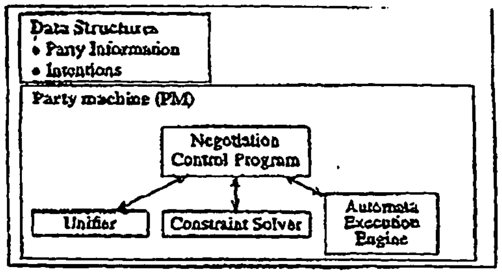


Figure 1. Party architecture

ordinator.

- The Constraints Solver is used to check sets of constraints.
- The Automata Execution Engine (AEE) executes commerce automata.
- The Unifier supervises the unification process.

The paper is organized as follows. Section 2 presents the basic terminology of EContracts. Section 3 defines intentions. In particular, it presents the commerce automata formalism and the way parties exchange messages during negotiations. Section 4 discusses unification, as well as "upgraded unification" that allows to perform unification by "relaxing" certain constraints. The EContracts unification algorithm is presented in an Appendix. The EC party system architecture is briefly discussed in Section 5. Section 6 presents related work. Conclusions are presented in Section 7.

2. Basics of the EContracts framework

The parties involved in an EC activity must agree on a common vocabulary. The "words" of this vocabulary are called classes and, formally, they are *rooted labeled ordered (RLO)* trees. The root of a class is labeled with the *class name*; the edges of the class are labeled with strings which hint at the function of the vertices; the leaves of the classes are labeled with typed variables.

Examples of classes are presented in Figure 2. We omit the type or the name of the variables when it is either irrelevant or clear from the context. For example, the Purchase contract class (Figure 2(a)) describes a used vehicle purchase transaction involving a customer, a used vehicle dealer, a list of purchased vehicles and a payment.

The presence of variables in a class enables its customization. There are four types of variables:

- *Atomic variables* The names of atomic variables begin with a \$ and the values that can be assigned to these variables are atomic values, i.e., string, real and integer.
- *Class variables* The names of class variables begin with a & and the values that can be assigned to these variables are class instances.
- *Atomic list variables* The names of atomic list variables begin with a % and the values that can be assigned to these variables are lists of atomic values.
- *Class list variables* The names of class list variables are enclosed between parentheses and the values that can be assigned to these variables are lists of class instances.

These notions are captured in the following definitions. We define the atomic types to be string, integer and real. Other types like date, boolean or enumerated types are possible. We limit ourselves to string, integer and real for the sake of simplicity. We assume the existence of a set of class names which is a set of strings.

Definition 2.1 A value *is either a string, an integer, a real, a class name or one of the special symbols* $\mathbb{N}$ *(for null),* $\mathbb{L}$ *(for lists) or* $\mathbb{CO}$ *for constraints.* □

Definition 2.2 A variable *is a triple* (t, n, v) *where* t *is an atomic type or a class name,* n *is a string and* v *is a value.* □

n is the name of the variable. A triple (t, n, v) must satisfy the naming constraints defined above (e.g., atomic variable names must begin with a \$ character), together with the obvious type-correctness constraint between t, n and v (i.e., a value must correspond to the type of the variable). A variable (t, n, v) is unbound if $v = \mathbb{N}$. A set of variables V is *proper* if every variable name in a triple of V is unique, i.e., appears in no other triple as the name component.

We define the words of the common vocabulary, namely the *classes*.

Definition 2.3 A class, *over a proper set of unbound variables VAR, is a rooted labeled ordered (RLO) tree, denoted* $(V, E, r, t, <_e, elf, vlf)$, *where*

- V *is a set of vertices,*
- E *is a set of edges,* $E \subseteq V \times V$,
- $r \in V$ *is the root of the tree,*
- t, *the label of the root, is a class name,*
- $<_e$ *is a partial order relation over* E *that defines the relative order of the edges that emanate from the same vertex,*
- $elf : E \rightarrow STRINGS$ *is the* edge labeling *function (defined so that the labels of the edges that emanate from the same vertex are all distinct), and*
- *Let* $V' \subseteq V$ *be the leaves of* T. $vlf : V' \rightarrow VAR$ *is the (total and onto)* leaf labeling *function.* □

In Figure 2, we represent the variable $(t, n, \mathbb{N})$ by t: n.

Classes are organized in *class hierarchies*, each defining a specialization hierarchy.

Definition 2.4 *Let* L *be a finite set of class names.* A class hierarchy H *over* L *is a directed labeled rooted tree in which every vertex is labeled with a class name in* L. *No class name may appear twice in the tree.* □

For example, Car is a specialization of Vehicle. As a consequence, Car instances can appear in the list of vehicles of a

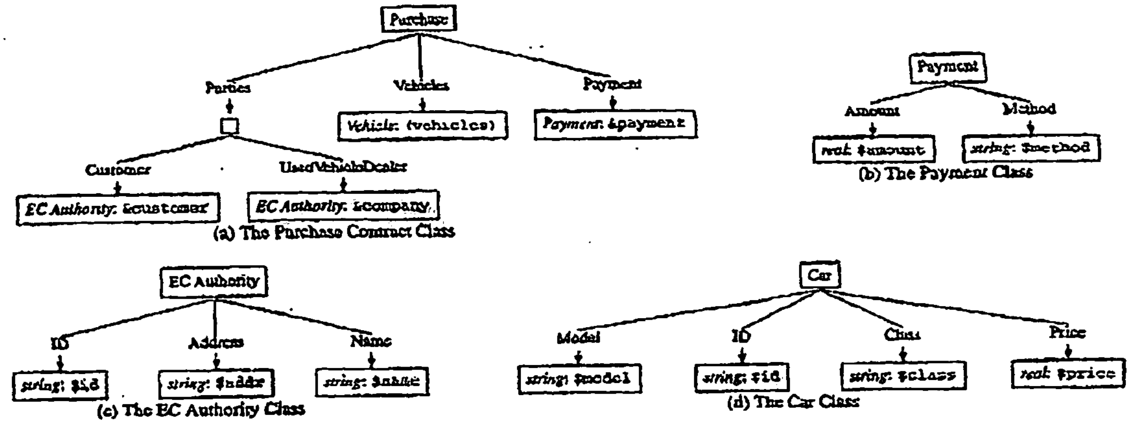


Figure 2. Examples of classes

Purchase contract. Such a relationship between classes does *not* imply any structural similarity between them.

An *ontology* is a set of hierarchies containing classes that are semantically related.

Definition 2.5 *Let $L_1, \ldots, L_n$ be pairwise disjoint sets of class names. An* ontology *over $L_1, \ldots, L_n$ is a set of class hierarchies $H_1, \ldots, H_n$ over $L_1, \ldots, L_n$, respectively*[1]. □

A class named a is *contained* in an ontology O if a is a vertex label in a class hierarchy in O. A class named b is a *child* of a class named a in O if (1) there exists a class hierarchy T in O such that T contains two vertices u and v labeled with a and b, respectively, and (2) there is an edge from u to v. Let the *descendant* relation be the reflexive and transitive closure of the child relation.

We assume the existence of three *basic* ontologies. The *contracts* ontology contains the possible EC contracts (e.g., Purchase, Rent). The *items* ontology contains goods and services (e.g., car, hair-cut) that can be the subjects of an EC activity. The *general* ontology contains EC general concepts (e.g., EC authority, payment, interest rate).

For each class name t defined in the basic ontologies, we assume the existence of a *canonical class* for t, denoted C_t, i.e., a class whose root is labeled with t. An *instance of type* t is a class that is isomorphic to C_t, i.e., identical up to a consistent renaming of variables.

A *party* is an active component that may be involved in an EC activity, for example an EC WWW site or a customer buying agent. The EContracts framework assumes a symmetric model, that is the structure of all parties involved in an EC activity is identical. A party manages the *party information*, i.e., a standard relational database that contains the party's global data, e.g., the party's identity, item lists, pricing information etc.

[1] When $L_1, \ldots, L_n$ are understood from context we will omit the phrase *over* $L_1, \ldots, L_n$.

3. Intentions

In addition to the party information, in order to be machine analyzable, a party should include a formal specification of the way it operates, i.e., the skeleton of contracts it may enter as well as the business rules and the constraints it enforces. The EContracts framework represents this information in *intentions*. Whereas classes are the "words" in our common language, *intentions* are the "sentences" of this language. Sentences are built by connecting words. An intention is composed of an *intention tree* which is derived from classes, *commerce automata* which encode business rules, and *constraints*.

3.1. Intention trees

Intention trees describe the structure of EContracts a party is willing to establish. Intention trees are derived from EC contract classes and are transformed into actual contracts by *instantiating* variables, in order to define the party's requirements, and by adding *operator vertices* that enable the specification of powerful logical constraints.

For example, in Figure 8[2], the customer, J. Smith, wishes to purchase two motorcycles or, alternatively, an Economy class car. He is ready to pay by either cash or check. In Figure 9, the PurchaseOnline Corp. is selling cars (one at a time) and is only accepting cash.

Note that the two intention trees in Figures 8 and 9 are complementary. The purpose of unification is to detect this fact and to build a "unified" tree from the intentions, namely the EContract.

[2] Since we refer several times, to the same intention trees, we have grouped their figures at the end of the paper.

Variable instantiation. Formally, variables are instantiated by using the α operator. The α operator takes a tree and a variable-instantiation operation, and produces a tree as follows. Let T be an intention tree and let $x = (t, n, \mathrm{N})$ be an unbound variable appearing in T.

- If x is an atomic variable, and v is a value of type t, $\alpha(T, x = v) \triangleq T'$ where T' is presented in Figure 3(a). In the figure, $\boxed{T}$ symbols represent (sub-)trees and $\boxed{X}$ symbols represent vertices.
- If x is a class variable, let t' be a class name which is a descendant of t in an ontology and let O'_2 be an instance of type t'. $\alpha(T, x = O'_2) \triangleq T'$ where T' is presented in Figure 3(b).
- If x is a class list variable, let $(t'_1, \ldots, t'_n)$ be a sequence of class names which are descendants of t in an ontology and let $(O'_1, \ldots, O'_n)$ be instances of types $(t'_1, \ldots, t'_n)$, respectively. $\alpha(T, x = (O'_1, \ldots, O'_n)) \triangleq T'$, where T' is is presented in Figure 3(c). Instantiation of atomic list variables is defined similarly.
- If x is a class list variable, let $(t'_1, \ldots, t'_n)$ be a sequence of class names which are descendants of t in an ontology and let $(O'_1, \ldots, O'_n)$ be instances of types $(t'_1, \ldots, t'_n)$, respectively. $\alpha(T, x \supseteq (O'_1, \ldots, O'_n)) \triangleq T'$, where T' is presented in Figure 3(d). The meaning is that the list that can be assigned to x must contain at least $(O'_1, \ldots, O'_n)$. We say that x must satisfy a *list containment constraint*. This operation is a partial assignment, as it constrains the possible values of x. List containment constraints on atomic list variables are defined similarly. A list containment constraint of the form $x \subseteq (O'_1, \ldots, O'_n)$ can be specified using OR vertices which are defined below.

Operator vertices. Operator vertices are vertices labeled with the strings "AND," "OR" and "NOT." Operator vertices are added to an intention tree by using the Δ operator. Let T be an intention tree, T' is *derived* by adding to T an operator vertex o as follows:

- o is an OR vertex or an AND vertex. Let u be a vertex in T, let (u, v) be an edge labeled with *lab*, and let V be the subtree rooted in v. Let V_1, V_2 be trees isomorphic to V up to renaming of variables. $\Delta(T, u, o, (V_1, V_2)) \triangleq T'$ where T' is obtained from O by (1) removing V from O, (2) adding an edge (u, o) labeled *lab*, and (3) adding edges from o to the roots of V_1 and V_2 (see Figure 4(a)).
- o is a NOT vertex. Let V_1 and V_2 be the two subtrees rooted at an AND vertex, such that their roots are not already NOT vertices. o can be added to the intention tree as the root of one of V_1, V_2, as described in Figure 4(b) (for V_2).

Figure 5 presents an example of the use of operator vertices. Recall that CO is a list containment constraint, e.g., variable `(x)`, when instantiated, should contain at least O_1 and O_2. The meaning is that the list of items (of type t) must contain at least O_1 and O_2, or O_4 and O_5, but must not contain O_3.

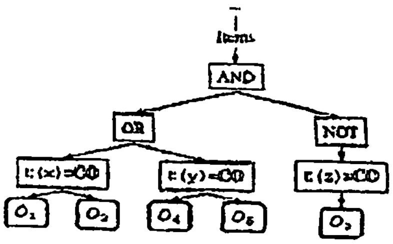


Figure 5. Using operator vertices

In an intention, we require that the (sub-)trees rooted at vertices that are labeled with the same variable name be identical.

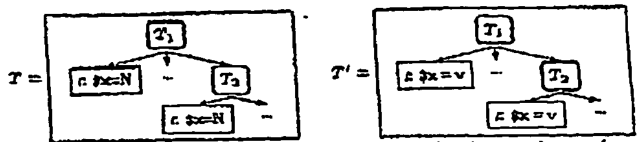
3.2. Constraints

An intention contains a set of *constraints*. A *constraint* is a function from a value assignment (to a set of variables) to the boolean values TRUE and FALSE. The sub-language used for the expression of constraints is not part of the EContracts framework specification. For the sake of simplicity, in examples, we use a simple constraints sub-language called *SIMPLE-C* which is presented through examples. For example, `not(Ground($title)) AND ($price > 100 ) AND ($name = ''John'') AND (($name, $price) ∈ R)` is a constraint. Note that `Ground` means "is not null" and `R` denotes a set (relation) of tuples. The assignment C = {`$title` ↦ *null*, `$price` ↦ 150, `$name` ↦ "John", `R` ↦ {("John",150),("Steve",170)}} *satisfies* the constraint.

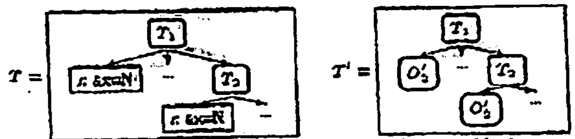
3.3. Commerce automata (CA)

Negotiations upon variable values (e.g., `$price`) and business rules enforcement can be expressed by intentions. Such intentions are built by assigning *commerce automata* to atomic variables, class variables and list variables that appear in intention trees. For the sake of simplicity, in this article, we do not consider automata for list variables. They are an extension of the automata for class variables. Furthermore, we consider business rules and negotiations involving two parties. However, it is possible to define negotiation protocols involving more than two parties.

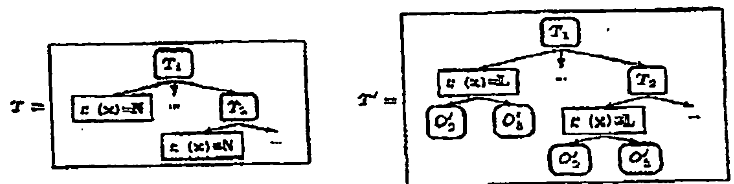
Variables. Consider a CA A which is assigned to a variable x of type t in an intention tree. If x is an atomic variable, *executing* A leads to the assignment of an atomic value of type t to x. In this case, the set of A's *output variables* is $\{x\}$. If x is a class variable, A specifies an *output instance* O of type t', where t' is a descendant of t in an ontology. Let $x_1, \ldots, x_n$ be the atomic variables that appear in O. Executing A assigns atomic values to some of the variables among $x_1, \ldots, x_n$ and assigns the resulting instance to x. In this case, the set of A's output variables is $\{x_1, \ldots, x_n\}$.



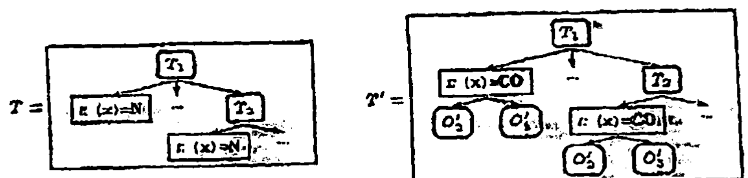
(a) T' is the tree resulting from the assignment of the atomic value v to the atomic variable \$x in T.



(b) T' is the tree resulting from the assignment of the instance O'_2 of type t' to the class variable &x in T. In T', the root of O'_2 is labeled with the variable (t; &x= t').

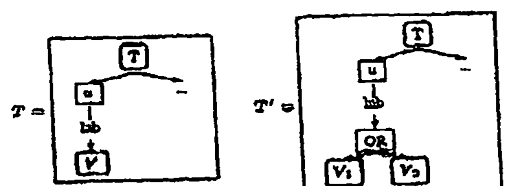


(c) T' is the tree resulting from the assignment of the list of instances (O'_2, O'_3) to the class list variable (x) in T.

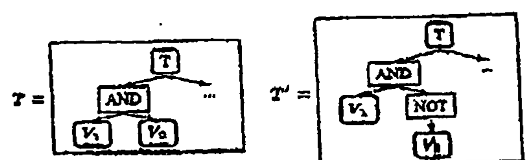


(d) T' is the tree resulting from the definition of the *list containment constraint* $(x) \supseteq (O'_2, O'_3)$ in T.

Figure 3. Variable instantiations



(a) T' is the result of adding an OR vertex to T — Note that V_1 and V_2 must be isomorphic to V up to renaming of variables. Adding an AND vertex is done in a similar way.



(b) T' is the result of adding a NOT vertex to T — Note that NOT vertices can be added only to subtrees rooted at an AND vertex.

Figure 4. Adding operator vertices

To build the assignment to the output variables, A uses a set of *internal variables* which can be atomic variables or relation variables, i.e., variables that can be assigned entire relations.

An automaton is provided with an initial assignment to its variables (determined by unification, see the example in Section 4) and the assignment may be modified during its execution. The atomic variables are typed and the relation variables have an associated arity.

Parties and messages. The execution of a CA is defined relative to two parties that exchange *messages*. The roles of the parties during the execution are asymmetric. The *active* party sends *inquiry messages* to the *passive* party, and the latter replies with *answer messages*.

The possible inquiry messages and the corresponding answer messages are:

- `Send $t`. The active party requests an assignment for the variable $t. The passive party replies with x, a value to assign to $t.
- `Confirm $t=v`. The active party requests a confirmation regarding the assignment of value v to $t. The passive party replies with either `Yes` or `No`.
- `Choose n from R`. The active party requests the passive party to choose n tuples from relation R. The passive party replies with a relation containing n tuples from R.
- `Change column c in R`. The active party requests the passive party to modify column c in relation R. The passive party replies with the modified relation.

The relational store. During its execution, a CA can query the relations in its *relational store* which contains (1) the relations in the active party's party information and (2) relations for relation variables. Furthermore, a CA can access the labels of vertices of intention trees. For example, the value of the Number leaf in the intention tree in Figure 8 is given by `Purchase.Parties.Customer-.Address.Number`.

States. A CA has a set of *states* S. One state is distinguished as the *starting state* and there exists a non-empty subset $S_f \subseteq S$ of *final states*. Each state is labeled with an *assignment program*, i.e., a sequence of assignment statements. Given an assignment to the automaton variables, say σ, the *execution* of an assignment program P modifies σ by executing the assignment statements, one after the other. The assignment statements and their semantics are presented via the example in Table 1.

The CA *transition function*, say δ, is applied to a state and a constraint and yields a state. For example, $\delta(s_1, \texttt{ground(\$x)}) = s_2$, means that if the CA is in state s_1 and the variable $x is ground (in the state of the current assignment) then the CA should move to state s_2.

Definition 3.1 (Commerce Automaton) *A commerce automaton, say A, is a tuple $A = (S, b, S_f, O, V, P, f_P, \delta)$ where:*

- *S is a set of states.*
- *$b \in S$ is the starting state.*
- *$S_f \subseteq S$ is the set of final states.*
- *O is the output instance.*
- *V is the set of the automaton variables.*
- *P is a set of assignment programs and f_p is a function that maps states in S to programs in P.*
- *δ is the (partial) transition function $\delta : S \times SC \longrightarrow S$, where SC is the set of all SIMPLE-C constraints.* □

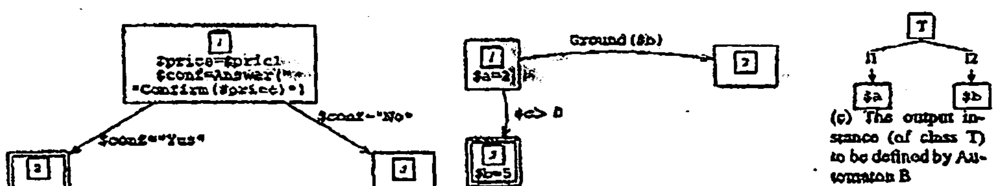
As part of a CA execution, messages are sent and answers are received. Formally, we model the sequence of received messages via a stack of messages. Given an initial assignment σ and a stack of answer messages Γ, the *execution* of the automaton is defined as follows. The execution begins at the starting state with the initial assignment. When the automaton enters a state s it modifies σ by executing the program $f_p(s)$. If $f_p(s)$ involves message exchanges, the answer to each inquiry message is popped from Γ. We assume that if Γ is empty or the answer message in Γ does not correspond to the inquiry message, then the execution stops. The constraints on the transitions from s are checked. If none is TRUE, or more than one is TRUE, then the execution stops. If exactly one is true, the automaton moves to the new state. If no transition exits from s, the execution stops in s. If the execution stops in a final state, then the execution is *successful*, and otherwise it *fails*.

Consider the CA APrice in Figure 6(a) which is assigned to the atomic variable `$price` in the used car dealer's intention (Figure 9). The company uses this automaton in order to assign a value to `$price`. The starting state is 1. First, the value of the variable `$pric1` (computed elsewhere as described in section 4) is assigned to `$price` and, using the `Answer` construct, the company asks the customer for price confirmation. The customer's answer is assigned to the variable `$conf`. If it is Yes, the automaton's execution is successful (state 2), otherwise it fails. It fails in state 1 (if the answer is neither Yes nor No) or in state 3 (if the answer is No).

Consider the CA in Figure 6(b) (An order condition involving a variable which is not ground evaluates to FALSE). This automaton is assigned to a class variable, say x, and, therefore, it defines an *output instance* (shown in Figure 6(c)) that should be assigned to x, in case the execution of the automaton succeeds. The automaton is provided with an initial assignment of its variables. If the initial assignment is {$b↦1}, the automaton enters state 2 and stops. Since 2 is not a final state, the execution fails. If the initial assignment is {$b↦1, $c↦1} the two constraints are satisfied, therefore the automaton cannot move to the next state, and, therefore, stops (and fails) in state 1. If the initial assignment is {$c↦1} the automaton enters state 3 and the execution is successful: 2 and 5 are assigned to $a and $b, respectively, in the output instance.

Instruction	Current Assignment
Initially	$\$a \mapsto 2$, $\$b \mapsto 3$, $R \mapsto \{(1,3),(2,4)\}$
`$b = 5`	$\$a \mapsto 2$, $\$b \mapsto 5$, $R \mapsto \{(1,3),(2,4)\}$
`$a = $b`	$\$a \mapsto 5$, $\$b \mapsto 5$, $R \mapsto \{(1,3),(2,4)\}$
`($a,$b) = Select * From R`	The values of the columns of the first tuple returned by the query are assigned to \$a and \$b. $\$a \mapsto 1$, $\$b \mapsto 3$, $R \mapsto \{(1,3),(2,4)\}$
`$a = Answer("Send $a")`	The active party sends the message "`Send $a`". The value returned by the passive party, say 8, is assigned to \$a. $\$a \mapsto 8$, $\$b \mapsto 3$, $R \mapsto \{(1,3),(2,4)\}$
`Q = Answer("Choose 1 from R")`	The active party sends the message "`Choose 1 from R`". The relation returned by the passive party is assigned to Q. $\$a \mapsto 8$, $\$b \mapsto 3$, $R \mapsto \{(1,3),(2,4),\}$, $Q \mapsto \{(2,4)\}$
`R = Answer("Change column 2 in Q")`	The active party sends the message "`Change column 2 in Q`". The relation returned by the passive party is assigned to R. $\$a \mapsto 8$, $\$b \mapsto 3$, $R \mapsto \{(2,5)\}$, $Q \mapsto \{(2,4)\}$

Table 1. An assignment program



(a) Automaton APrice

(b) Automaton B

(c) The output instance (of class T) to be defined by Automaton B

Figure 6. Commerce automata - The final states have double frames

We now formally define an intention and a party data structure. An *intention* is a tuple (T, F, S, C) where T is an intention tree, S is a set of CAs, F is a partial function from the atomic variables and the class variables that appear in T to S, and C is a set of constraints. A party data structure is a tuple (SI, I) where SI is the party information and I is a set of intentions indicating the EContracts the party is ready to enter.

4. Unifying two intentions

4.1. Basic unification

The unification algorithm is an extension of the unification algorithm for terms in logic (for example, see [13]). The algorithm is extended to handle operator vertices and CAs and is presented through the following example (the algorithm is formally described in the Appendix). The customer (whose intention tree is shown in Figure 8) has one constraint, i.e., that the cost is less than \$300 (`$price` < 300). The used car dealer (whose intention tree is shown in Figure 9) has two CAs. The Acar CA, which is associated with the variable `&Car` in the intention tree, describes how cars are sold (Figure 7 contains a part of the automaton definition). The second CA, APrice (shown in Figure 6(a)), is associated with the variable `$price`. PurchaseOnline's site information consists of one relation Rcar(Model, ID, Class, Price).

The unification starts at the roots of the two intention trees. In the `Parties` subtree, the `Customer` and `UsedVehicleDealer` edges are unified. Corresponding subtrees are assigned to the variable `&customer` and `&company`, respectively. At the `Vehicles` subtree, the algorithm reaches an OR vertex in the customer's intention tree. In turn, each subtree under the OR vertex is unified with the `Vehicle` subtree in the company's intention tree. The unification of the left branch obviously fails (since, it is impossible to unify a list of two motorcycles with a list that contains one car). At the right branch under the OR vertex of the customer's intention tree, the vertex labeled `Car` is unified with the vertex labeled `&Car` in the company's intention tree. Since the variable `&Car` is assigned to the Acar CA (shown in Figure 7(b)), the customer's subtree rooted in `Car` is unified with the output instance of the Acar CA (shown in Figure 7(a)). This unification provides the *initial* assignment for the CA variables, i.e., Economy is assigned to `$class`.

The Acar CA is executed as follows. Since the variable `$class` is ground, the automaton moves to state 1. The automaton assigns to the relation variable A all the cars that correspond to the customer's class specification (Economy). Then, the automaton asks the customer to choose a model. This choice may be done in several ways; human intervention may be requested, or an automatic tool may be used. Such an automatic "expert" tool may, simply, choose an arbitrary car or employ more complex "user defined" strategies. It can also try every choice, one after the other, and use backtracking if some choice leads to the failure of the automaton execution. After receiving the model name, the automaton selects the car, say (Cavalier, 322, Economy, 230), from the database (state 2).

Following this assignment, the current constraint set ({`$pric1` = 230, `$price` < 300}) is verified as satisfiable and the unification algorithm resumes.

The unification proceeds to the `Amount` edge and the APrice CA (shown in Figure 6(a)) is executed. After confirmation, the new set of constraints {`$pric1` = 230, `$price` = 230, `$price` < 300} is checked and the unification proceeds to treat the Payment subtree. The generated EContract is depicted in Figure 10.

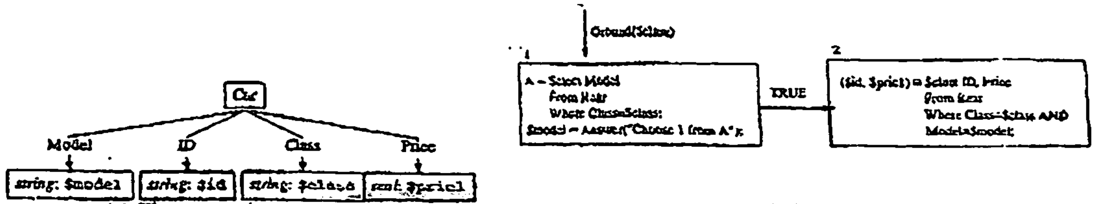
4.2. A Best Effort Unification Algorithm

Here we address situations in which the unification algorithm fails. We propose an approximate unification algorithm. The basic underlying idea is that of *upgrading* parts of intention trees. As an example, suppose that an intention tree specifies the constant *red* in a node. Suppose further that this is in fact a preference rather than an ultimate requirement. One option is to use OR nodes and give additional color options (observe, that by the way the unification algorithm operates, priorities of colors are naturally assigned in left to right order). But, it's tedious to specify many colors, in addition the party may even allow *any* color as a last resort. The solution we propose is to *prioritize* the (some of) the nodes and edges of the intention tree. A priority is a natural number, the higher the number, the more important is the constraint represented by the node. Similarly, constraints in the set of constraints associated with the intention may also be prioritized.

If pririties are assigned to edges connected to the alternatives of an OR node, AND node, or to list elements, then the highest priority alternatives within such OR, AND or list node are tried first (think of the others as temporarily eliminated). So, these priorities indicate an order for the unification algorithm. (This is not coded in the algorithm presented in the Appendix.)

Priorities in nodes and edges can be used as follows. Suppose unification fails, one can take a low priority node and replace it with a "less constraining" one. This can be done by "upgrading" the node. An *node upgrade* can be any sequence of *actions* applied to nodes in the intention which result in a legal tree. Some actions apply to edges and make them less constraining. An *action* is defined as one of:

1. Consider a node labeled with a variable of class c, the variable is modified to be of class *csup* which is a



(a) The output instance (b) The CA

Figure 7. The Acar CA

superclass of class c. The modification applies to all occurrences of the variable.

2. Consider a node labeled with a variable X. The node is labeled with a variable Y. If variable Y appears elsewhere in the intention, the node must be labeled with the same class as in other occurrences. Variable Y may also be a "fresh new variable".

3. Consider a node which is the root of a subtree and labeled by a (bound) variable. Make the variable unbound by erasing the subtree, except for the node that is now labeled with the unbound variable. The modification applies to all occurrences of the variable.

4. Consider a list node connected via edges to the list elements. Eliminate a list element.

5. Consider a node labeled with a variable X which is associated with automata (one or more). Change the set of associated automata, in particular disassociation with automata altogether is a possibility.

6. Consider an edge e with label lab. Change lab to another label that appears in one of the intentions or eliminate the label altogether.

7. Consider node in the tree, connected via an edge e to its parent, which is the root of a subtree T. Eliminate edge e and the subtree T altogether. This is a drastic measure that may be neede if, for example, multiple versions of classes exist due to out of date or erroneous software. It is also possible that portions of trees are stored in various media or locations that may be inaccessible, temporarily or permanently.

So, upgrading provides further options to the unification algorithm to explore. It may result in solutions that only "approximate" true unification of the pre-upgraded intentions. Thus, this gives meaning to "unify as much as possible".

There are some technical issues to consider. One is that the upgrading of the least "important" node may not suffice to produce a unified intention. Further upgrades may be required. The issue is in what order to perform them. Suppose that 4 nodes: A,B,C and D are labeled with decreasing priorities, say 4,3,2 and 1, respectively. An order that seems most reasonable is as follows (other orders are also possible):

1. upgrade D;
2. if above fails upgrade C;
3. if above fails, upgrade C and D;
4. if above fails, upgrade B;
5. if above fails, upgrade B and D;
6. if above fails, upgrade B and D and C;
7. if above fails, upgrade A;
8. if above fails, upgrade A and D;
9. if above fails, upgrade A and D and C;
10. if above fails, upgrade A and D and C and B;

We note that one may demand an aggregate condition in defining the "best approximation", e.g., as a function of priorities as well as the number of upgrades. Observe that if a prioritized node u is the parent of another node v, once v is labeled with an unbound variable there is no need to try upgrades to node v, the node is no longer in the tree.

Another issue is that we are given two intentions, in what order should we apply the upgrades to these two intentions? A reasonable way is to alternate between the two intentions. One way to implement this is to map the priorities of the first intention to odd numbers and those of the second to even numbers, and then apply them in priority order as described above. Other prioritizing schemes are also possible.

Constraints may be relaxed in a similar way. Here we may have more degrees of relaxation. For example, $a < b$ may be relaxed to $a \leq b$. Relaxation may also eliminate a constraint altogether. Here again, the priority of the constraint indicates its importance and order of relaxation or elimination.

5. The EContracts execution modules

The execution modules are the *unifier*, which executes the unification algorithm, the *constraint solver*, which ensures the satisfiability of constraint sets generated during unification and the *automata execution engine*, which is responsible for the execution of commerce automata. All these modules are organized and activated when needed by the *negotiation control program* (NCP) (see Figure 1).

- **The Constraints Solver** When needed, the NCP delivers a set of constraints to the constraints solver. The constraints solver returns to the NCP an answer which may be either `Unsatisfiable`, i.e., it is impossible to find an assignment to the variables that appear in the constraints such that the constraints are satisfied, or `Satisfiable`, i.e., there exists a satisfying assignment. If the constraints solver returns `Satisfiable`, it may return a modified (and simplified) constraints set.
- **The Automata Execution Engine (AEE)** When needed, the NCP requests the AEE to execute an automaton. The NCP and the AEE may not belong to the same party. For example, when unifying the intentions of Party A and Party B, Party A's NCP may request from Party B's NCP to forward a request of an automaton execution to Party B's AEE. When the execution ends, the AEE returns either `SUCCESS`, i.e., the CA reached a final state, or `FAILURE`, i.e., the CA did not reach a final state. If the AEE returns `SUCCESS`, the NCP may modify the EContract by utilizing the CA's output.
- **The Unifier** executes the unification algorithm on two intentions submitted by the NCP. If it succeeds, the unifier module returns the EContracts. The unifier module may occasionally request the NCP to pass a set of constraints to the constraint solver or to pass a CA to the AEE for execution (the AEE may be the AEE of either Party A or Party B).
- **The Negotiation Control Program** is the main execution module. The NCP manages the negotiation process between the parties and coordinates the activation of the other execution modules.

6. Related work

We do not intend to give an exhaustive overview of research in EC. References regarding authentication and payment can be found in [18]. References to a game-theoretic approach to automated negotiation can be found in [16]. Some applications of agents technology are related to EC [14], e.g., information-brokers [7] and Kasbah [6]. Agent systems are mostly task oriented and do not define a general framework for negotiation. Usually, the agents' business model is encoded in its program and therefore, practically, cannot be analyzed. The W3C EC interest group (*www.w3.org*) is working towards standards in the domains of micropayment, security, and public policy. The combination of XML and EDI is discussed in [21].

The Eco system [17] implements an architecture for Internet commerce. Eco has some limitations in comparison to EContracts. Messages and knowledge bases are written in high level analyzable languages (KQML [12] and KIF [9]). However, agent behavior is described using common programming languages which are difficult to analyze.

Important concepts of the EContracts framework are class, class hierarchy and ontology. Our framework is similar to the approaches of [20, 15]. Commerce automata have roots in the works of [1] and [4].

The execution modules benefit from extensive research in constraint programming [10, 19] and logic programming [13]. In particular, unification of class structures is described in [3].

Relational databases are also used as a foundation for EC. In [8], an active database model is used to capture the semantics of the constantly changing EC environment. In [2], business models are formalized using *relational transducers*. In [5], ideas similar to ours are presented; however, a complete framework is not described. A preliminary version of this work has been presented in [11].

7. Conclusions and future work

The EContracts framework enables (semi-)automated EC. The framework defines standard data structures and execution modules that comprise an autonomous party.

We are extending our model to support *open EContracts*, i.e., contracts in which a deal is not totally specified when the EContracts is signed. In such an EContract, it is possible to encode, for example, that the model of the purchased car is open and will be fixed by the customer while the final price is between $250 and $300 and will be fixed ultimately by the used car dealer.

We plan to develop a practical marketplace based on EContracts. We also plan to work on theoretical problems such as deal feasibility, analysis of negotiation strategies and optimization of negotiation behaviors.

References

[1] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 209–219, 1991.

[2] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Proc. PODS*, pages 179–187, 1998.

[3] H. Ait-Kaci, A. Podelski, and S. C. Goldstein. Order sorted feature theory unification. *JLP*, 30:99–124, 1997.

[4] L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: Np-

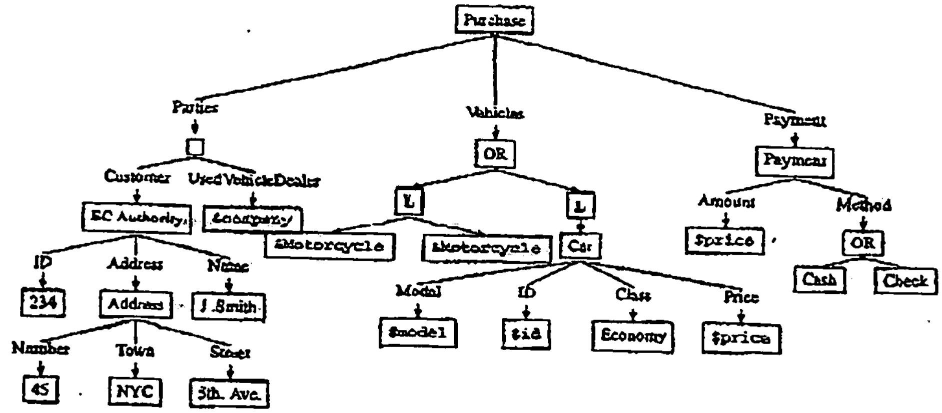


Figure 8. The customer's intention tree

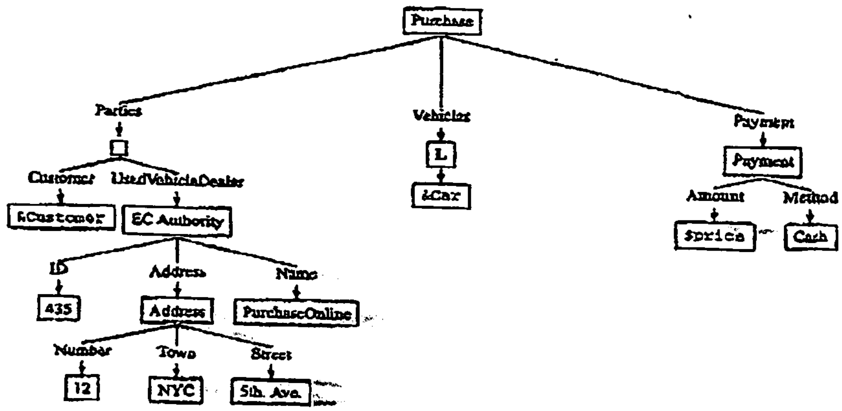


Figure 9. The used car dealer's intention tree

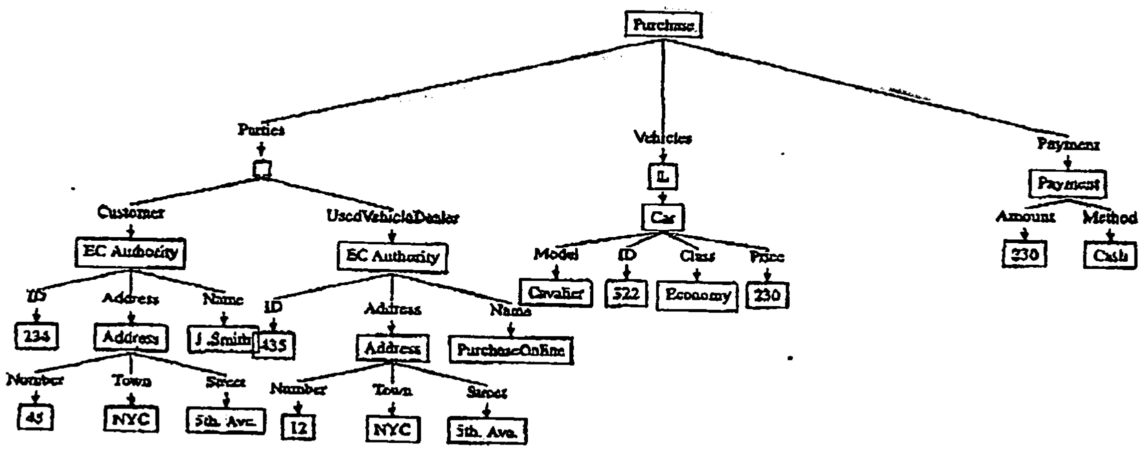


Figure 10. The generated Econtract

completeness, recursive functions and universal machines. *Bulletin of the AMS*, 21(1):1–46, July 1989.

[5] S. Boucher. Open nested type structures and partial unification for searching in distributed electronic market. In *Proc. TrEC'98*, 1998.

[6] A. Chavez and P. Maes. Kasbah: An agent marketplace for buying and selling goods. In *Proceedings of the First International Conference on the practical Application of Intelligent Agents and Multi-Agent Technology*, London, UK, 1996.

[7] R. Fikes, R. Engelmore, A. Farquhar, and W. Pratt. Network-based information brokers, 1995. http://www.ksl.stanford.edu/KSL_Abstracts/KSL9513.html.

[8] B. Fordham, S. Abiteboul, and Y. Yesha. Evolving databases: An application to electronic commerce. In *IDEAS*, pages 191–200, 1997.

[9] R. Genesereth. Knowledge interchange format. In *Proceedings of the Conference of of the Principles of Knowledge Representation and Reasoning*, pages 599–600, 1991.

[10] D. Kapur. *Principles and Practice of Constraint Programming*, chapter An Approach for Solving Systems of Parametric Polynomial Equation, pages 217–243. The MIT Press, 1995.

[11] D. Konopnicki, L. Leiba, O. Shmueli, and Y. Sagiv. Toward automated electronic commerce. In *First IAC Workshop on Internet-Based Negotiation Technologies*. IBM TJ Watson Research Center, Yorktown Heights, NY, 1999.

[12] Kqml: A knowledge query and manipulation language. KQML Advisory Group, http://www.cs.umbc.edu/kqml.

[13] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[14] Agent research projects. http://lcs.www.media.mit.edu/groups/agents/projects.

[15] Ontolingua. http://ontolingua.stanford.edu.

[16] J. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiations among Computers*. MIT Press, 1995.

[17] J. M. Tenenbaum, T. S. Chowdhry, and K. Hughes. Eco system: An internet commerce architecture. *Computer*, 30(5):48–55, May 1997.

[18] J. D. Tygar. Atomicity versus anonymity: Distributed transactions for electronic commerce. In *Proc. VLDB*, pages 1–12, 1998.

[19] T. Uribe. Sorted unification using set constraints. *CADE*, pages 163–177, 1992.

[20] M. Uschold, M. King, S. Moralee, and Y. Zorgios. The enterprise ontology, 1995. http://www.aiai.ed.ac.uk/ enterprise/enterprise.

[21] Introducing xml/edi-edifact. http://www.w3.org/ECommerce/overview/xmledifact.

A. The unification algorithm

The input of the unification algorithm is two intention trees I_1 and I_2. C is the set of global constraints defined on I_1 and I_2. We assume that I_1 and I_2 do not contain OR vertices (we can remove all the OR vertices from an intention tree as described informally in Figure 11).

The algorithm is described in a Prolog-like language. We assume the existence of the following predicates:

$instance(T, S)$ Generates an isomorphic copy S of T with a disjoint set of variables (including association to automata).

$var(V)$ Satisfied if V is an unbound variable.

$nonvar(V)$ Satisfied if V is not an unbound variable.

$atomic(V)$ Satisfied if V is an atomic variable.

$class(V)$ Satisfied if V is a class variable.

$int(V), float(V), string(V)$ Satisfied if V is ground and is an integer (float, string, resp.).

$mode(V, Structure, Type)$ Satisfied if variable V has the structure $Structure$ ($Structure$ can be: atomic, class or list) and type $Type$ ($Type$ can be: int, float, string or a class name).

$bassert(Edb)$ Backtractable assertion of the fact Edb.

$bretract(Edb)$ Backtractable retract of the fact Edb.

$classDesc(C_1, C_2)$ Class C_1 is a descendant of class C_2 in a class hierarchy of an ontology.

$automaton(V, A, O, ModeStmt)$ Satisfied if automaton A is assigned to V, its output instance is O and O's mode statement is $ModeStmt$.

$run(A, O)$ Execute automaton A on instance O.

$checkConstraints(T, C)$ Satisfied if the variables in T satisfy the constraints set C.

$permute([X_1, \ldots X_n], [Y_1, \ldots Y_n])$ Satisfied if $(Y_1, \ldots Y_n)$ is a permutation of $(X_1, \ldots X_n)$. Backtracking generates the "next" permutation.

The Unification Algorithm follows:

Unification Algorithm

/* Main */

/* We assume that a fact (clause) is asserted prior to running the unification, for each of the variables in the input intentions I_1 and I_2, in the following way: $assert(mode(variable\ name, structure, type))$.*/

/* Run Unification; the result is I_1. */

$main\_unification(I_1, I_2) \leftarrow unify(I_1, I_2),$
$checkConstraints(I_1, C), write(I_1).$

/* Use symmetry. */

$unify(I_1, I_2) \leftarrow unify1(I_1, I_2).$
$unify(I_1, I_2) \leftarrow unify1(I_2, I_1).$

/* Unifying 2 Classes. */

$unify1(T_1, T_2) \leftarrow nonvar(T_1), nonvar(T_2),$
$T_1 = CN_1(E_1 - C_1, \ldots, E_k - C_k),$
/* CN_i is class name, E_i is edge label to subtree C_i. */
$T_2 = CN_2(F_1 - D_1, \ldots, F_k - D_k),$
$class(CN_1), class(CN_2),$
$CN_1 = CN_2, E_1 = F_1, \ldots, E_k = F_k,$
$unify(C_1, D_1), \ldots, unify(C_k, D_k).$

/* Unifying 2 Lists. */

$unify1(T_1, T_2) \leftarrow nonvar(T_1), nonvar(T_2),$
$T_1 = \mathbb{L}(C_1, \ldots, C_k), T_2 = \mathbb{L}(D_1, \ldots, D_k),$

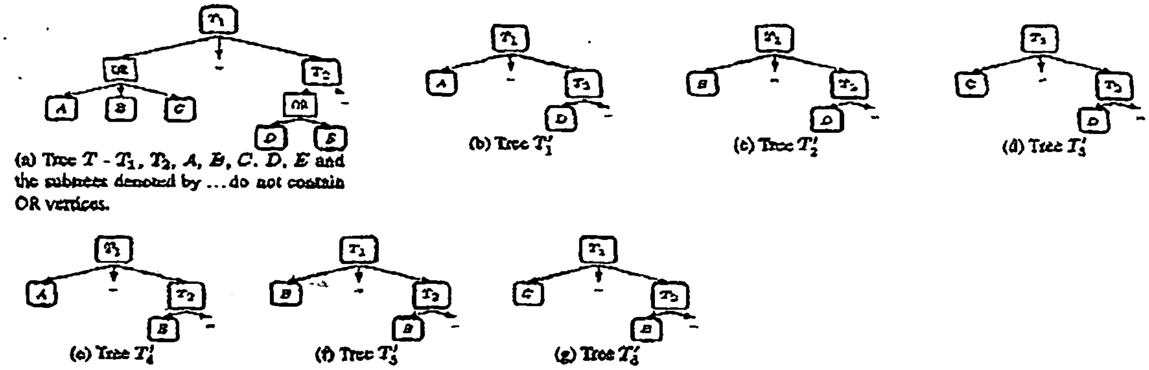


Figure 11. Removing OR vertices: an informal explanation

$permute([C_1, \ldots, C_k], [X_1, \ldots, X_k])$,
$unify(X_1, D_1), \ldots, unify(X_k, D_k)$.
/* Unifying a List and a superset constraint.*/
$unify1(T_1, T_2) \leftarrow nonvar(T_1), nonvar(T_2)$,
$T_1 = \mathbb{L}(C_1, \ldots, C_k), T_2 = \mathbb{CO}(D_1, \ldots, D_m)$,
$m < k$, /* T_1's elements must include T_2's. */
$permute([C_1, \ldots, C_k], [X_1, \ldots, X_k])$,
$unify(D_1, X_1), \ldots, unify(D_m, X_m)$.
/* Both T_1 and T_2 are rooted at a NOT vertex. */
/* There is no need to unify T_1 and T_2.*/
$unify1(T_1, T_2) \leftarrow nonvar(T_1), nonvar(T_2)$,
$T_1 = NOT(ST_1)$,
$T_2 = NOT(ST_2)$,
!.
/* T_1 is rooted at a NOT vertex. */
$unify1(T_1, T_2) \leftarrow nonvar(T_1)$,
$T_1 = NOT(ST_1)$,
$not(unify(ST_1, T_2))$.
/* T_1 is rooted at an AND vertex */
$unify1(T_1, T_2) \leftarrow nonvar(T_1)$,
$T_1 = AND(ST_1, ST_2)$,
$unify(ST_1, T_2), unify(ST_2, T_2)$.
/* We only list rules for integer constants and variables. Similar rules apply for the float and string atomic data types. */
/* 2 constants */
$unify1(T_1, T_2) \leftarrow$
$int(T_1), int(T_2), T_1 = T_2$.
/* Assigning a constant to a variable. */
$unify1(T_1, T_2) \leftarrow var(T_1)$,
$mode(T_1, atomic, int), int(T_2)$,
$bretract(mode(T_1, atomic, int))$,
$T_1 = T_2$.
/* 2 atomic variables. */
$unify1(T_1, T_2) \leftarrow var(T_1), var(T_2)$,
$mode(T_1, atomic, int)$,
$mode(T_2, atomic, int)$,
$bretract(mode(T_2, atomic, int))$,
$T_1 = T_2$.
/* Assigning a class instance to a class variable. */
$unify1(T_1, T_2) \leftarrow var(T_1), nonvar(T_2)$,
$mode(T_1, class, X)$,
$T_2 = CN(E_1 - C_1, \ldots, E_k - C_k), class(CN)$,
$classDesc(CN, X)$, /* CN is subclass of X. */
$bretract(mode(T_1, class, X))$,
$T_1 = T_2$.
/* 2 class variables. */
$unify1(T_1, T_2) \leftarrow var(T_1), var(T_2)$,
$mode(T_1, class, X)$,
$mode(T_2, class, Y)$,
$classDesc(Y, X)$,
$bretract(mode(T_1, class, X))$,
$T_1 = T_2$.
/* Assigning a list to a list variable. */
$unify1(T_1, T_2) \leftarrow var(T_1), nonvar(T_2)$,
$mode(T_1, list, X)$,
$T_2 = \mathbb{L}(C_1, \ldots, C_k)$,
/* Need to check whether the list variable and the list's items have compatible types */
$bassert(mode(A_1, \_, X)), \ldots$,
$bassert(mode(A_k, \_, X))$,
$unify(A_1, C_1), \ldots, unify(A_k, C_k)$,
$bretract(mode(T_1, list, X))$
$T_1 = T_2$.
/* A list variable and a superset constraint. */
$unify1(T_1, T_2) \leftarrow var(T_1), nonvar(T_2)$,
$mode(T_1, list, X)$,
$T_2 = \mathbb{CO}(C_1, \ldots, C_k)$,
/* No unification is done, the constraint is noted */
$bassert(constraint(\mathbb{CO}, T_1, T_2))$.
/* 2 list variables. */
$unify1(T_1, T_2) \leftarrow var(T_1), var(T_2)$,

$mode(T_1, list, X)$,
$mode(T_2, list, Y)$,
$classDesc(Y, X)$,
$bretract(mode(T_1, class, X))$,
$T_1 = T_3$.
/* Unifying an automaton variable. */
$unify1(T_1, T_2) \leftarrow$
$automaton(T_1, A, O, ModeStmt), bassert(ModeStmt)$,
/* Need a copy of T_2 because automaton execution is possibly at a remote site — different environment.*/
$instance(T_2, I), unify(O, I)$, /* Prepare automaton "inputs" */
$run(A, O), unify(O, T_1), unify(O, T_2)$.
/* Special case: unification of 2 superset constraints. */
$unify1(T_1, T_2) \leftarrow nonvar(T_1), nonvar(T_2)$,
$T_1 = CO(C_1, \ldots, C_k)$.
$T_2 = CO(D_1, \ldots, D_m)$,
/* No unification is done, constraints are noted. */
$bassert(constraint(CO, T_1, T_2))$

Note that if unbound variables appear in a subtree rooted at a NOT vertex, it is possible for the unification algorithm to fail while reaching an agreement was in fact possible. This problem arises because the unification order is fixed by prolog and does not take into account that unification with subtrees rooted at NOT vertices may be done after all the relevant variable bindings are determined. In such a case, it is possible to modify the intention trees so that the standard order of unification leads to a correct result. However, details are beyond the scope of this paper.

Glossary

Computational device: a device used to perform computations, it encompasses hardware (e.g. mechanical, electronic, chemical or biological) and software (programs, executables, byte-codes, systems, servers and the like) as well as a computer system or part of a computer system, a database, a server, or a communication system.
EC Party: A legal entity that may be involved in a deal. In particular, it can designate individuals, corporations, countries, state and local authorities, organizations and associations.
Intention: A specification of constraints to be satisfied by a deal. By a specific example said constraints designate the objectives of the deal (e.g., buy, rent), parties and objects involved in the deal, and limitations that are applied to these entities.
Component: A component is an entity that is a building block for intentions.
Atomic component: A component describing a simple entity such as a bit, a number or a string.
Compound component: A component that is built of other components.
Constraint component: A component describing constraints on other components, e.g. that one atomic component is larger than another.
Basic component: A component whose structure is known to a user community and is agreed upon as representing a real life concept. Basic components are named.
Variable component: A component that is represented by a variable.
Computable variable component: A variable component that is associated with one or more computational devices. Such a device transforms that variable into a component. This component usually includes further elaboration on the deal.
Fitting: A process of taking one or more intentions and reconciling them into intentions that together satisfy as much as possible the constraints prescribed by the original set of intentions.
Contract: A set of intentions that are agreed upon by the issuing parties. In particular, if that set consisting of one intention it's called a **simple contract**. If it includes no variable components it is called a **ground contract**.
Atomic value: a concrete representation of an atomic component.
Atomic type: A set of atomic values.
Class: a prototype of a compound component, for example a class in Java or C++, a compound term in logic programming (Prolog), a list structure in LISP, etc. The specification of a class can involve atomic types, values and classes. A class usually has a name.
Class value: a particular instance of a class prototype.
Basic class: A class that constitutes a basic component.
Variable: An entity with a name, a type (atomic, class, atomic collection, class collection where a collection is, e.g., list, set, subset, superset, one-of, array) and value (either atomic value, class value, undefined (called null), or a collection of values.)
Computable variable: A variable that is specified to be a computable component.
Abstract class: A class that has a name but no class instances. It used to abstract classes that appear a class hierarchy (see below).
Sub-class relationship: A statement that a class, say A, is more general than a class, say B. Classes A and B need not have similar prototypes. Classes may be abstract.
Class hierarchy: A collection of sub-class relationships. It is sometimes required that this relationship be transitively non-cyclic, i.e., that a class is not its own subclass.

Ontology: A collection of class hierarchies. It is sometimes required that no class name appears in more than one hierarchy of the ontology.
Item ontology: A particular ontology that usually contains names of basic classes that correspond to objects or concepts, for example car, bank account, John, Pepsi Co..
General ontology: A particular ontology that usually contains names of basic classes that correspond to transactions, for example buy, rent, lease, transport, invest, destroy, build.
Party information: A set of information items an EC party maintains. This set usually contains its identity, a collection of intentions, and other data relevant to its operation. The party information may change dynamically over time. Parts of it may be published for outsiders to access/view and parts of it may be restricted in terms of who and when can access/view them.
Operator class: A class that indicates constraints on values. Examples are OR, AND, NOT and ONE-OF.
Intention trees: An intention built by the following process. One starts with an instance of a general class and then may extend it via zero or more extension steps.
Extension step: An extension step is performed by: replacing a null atomic variable by an atomic value, or by replacing a null class variable with a new fresh copy of a class prototype, or by replacing a null collection variable by a collection of values, or by introducing an operator class instance and modifying the intention in accordance to rules of introduction of operator classes.
Constraint: A constraint component specifying limitations on values variables may be associated with, on relationships concerning variables and values, and on aggregates of values.
Message: Communication of information between one or more computational devices.
Reply: A message that is sent as a response to another message or messages.
Relation: A form of representing a collection of information items, each item is composed of fields and values for these fields.
Commerce automaton: A computational device that is specified using states, transitions among states, predicates on transitions, actions to be performed in a state, the forms of input, the forms of output. In particular, actions may be the sending or receiving of messages and creation/destruction/access/modification of values of variables including variables associated with relations and other relations (e.g., those associated with party information). A computed variable component is associated with one or more commerce automata.
Unification of intention trees: The fitting of intention trees. The process involves matching of similar components, the assignment of values to variables, the execution and/or analysis of commerce automata, exchange of messages. The result is one or more intention trees that satisfy as much as possible the constraints expressed by the original intention trees. If the parties, that present the original intention trees, agree upon the result, an **electronic contract (EContract)** is said to result. The EContract is **ground** if all variables are assigned non-null values The EContract variables may be associated with party or parties that are to determine their actual values at the point of execution of deal(s). . The EContract is single if it consists of a single intention.

Party machine (PM): A computational entity employed by an EC party which controls the process of fitting of intentions. The PM may communicate with other computational devices, and in particular other PMs, in attaining its mission. The major components of a PM are:

1. A Negotiation Control Program (NCP) is an overall coordinator.
2. A Constraints Solver is used to check, simplify, solve and evaluate sets of constraints.
3. An Automata Execution Engine (AEE) executes commerce automata.
4. The Unifier supervises the unification process.

CLAIMS:

1. An automatic electronic commerce system, comprising:

(a) plurality of parties, each including at least one intention that specifies deal constraints that said party is ready to engage; each intention contains at least one component and at least one party of said plurality contains at least one variable component associated with at least one computational device, which when activated at a given state transforms said variable component into a component.

(b) Party machine (PM) that includes at least a fitting module capable of fitting at least two intentions, so as to produce a deal that meets as much as possible the deal constraints of said at least two intentions; said fitting includes matching components in one intention from among said at least two intentions with counterpart components in another intention from among said at least two intentions; for any variable component from among the components stipulated in said (a), activating the computational device associated with said variable component in order to transform it to a component as part of applying said fitting.

2. For use in the system of Claim 1, a computational device.

3. For use in the system of Claim 1, a fitting module.

4. For use with the system of Claim 1, a matching component.

5. An automatic electronic commerce method, comprising the steps of:

(a) providing a plurality of parties, each including at least one intention that specifies deal constraints that said party is ready to engage; each intention contains at least one component and at least one party of said plurality contains at least one variable component associated with at least one computational device, which when activated at a given state transforms said variable component into a component.

(b) fitting at least two intentions, so as to produce a deal that meets as much as possible

the deal constraints of said at least two intentions; said fitting includes matching components in one intention from among said at least two intentions with counterpart components in another intention from among said at least two intentions; for any variable component from among the components stipulated in said (a), activating the computational device associated with said variable component in order to transform it to a component as part of applying said fitting.

6. Memory media containing at least one computer executable program capable of

(a) identifying a plurality of parties, each including at least one intention that specifies deal constraints that said party is ready to engage; each intention contains at least one component and at least one party of said plurality contains at least one variable component associated with at least one computational device, which when activated at a given state transforms said variable component into a component.

(b) fitting at least two intentions, so as to produce a deal that meets as much as possible the deal constraints of said at least two intentions; said fitting includes matching components in one intention from among said at least two intentions with counterpart components in another intention from among said at least two intentions; for any variable component from among the components stipulated in said (a), activating the computational device associated with said variable component in order to transform it to a component as part of applying said fitting.